Creating HTML Reports in PowerShell

>

PowerShell.org

Table of Contents

ReadMe	0
Introduction	1
HTML Report Basics	2
Gathering the Information	3
Building the HTML	4
Combining HTML Reports and a GUI Application	5
Contacting Me	6

Learn to properly use ConvertTo-HTML to produce multi-section, well-formed HTML reports – but then go further with a custom EnhancedHTML module! Produce beautiful, color-coded, dynamic, multi-section reports easily and quickly. By Don Jones.

Creating HTML Reports in PowerShell

By Don Jones

Cover design by Nathan Vonnahme

Learn to properly use ConvertTo-HTML to produce multi-section, well-formed HTML reports – but then go further with a custom EnhancedHTML module! Produce beautiful, color-coded, dynamic, multi-section reports easily and quickly. By Don Jones.

This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file as widely as possible, but ask that you do not modify the document.

Getting the Code The EnhancedHTML2 module mentioned in this book can be found in the https://www.powershellgallery.com/packages/EnhancedHTML2/. That page includes download instructions. PowerShellGet is requires, and can be obtained from PowerShellGallery.com

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing

process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

HTML Report Basics

First, understand that PowerShell isn't limited to creating reports in HTML. But I like HTML because it's flexible, can be easily e-mailed, and can be more easily made to look pretty than a plain-text report. But before you dive in, you do need to know a bit about how HTML works.

An HTML page is just a plain text file, looking something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  "http://www.w3.org/TR/xhtml1/DT
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>HTML TABLE</title>
</head><body>
<table>
<colgroup><col/><col/><col/><col/><col/></colgroup>
<tr><th>ComputerName</th><th>Drive</th><th>Free(GB)</th><th>Free(%)</th><th>Size(GB)</th>
<tr><td>CLIENT</td><td>C:</td><td>49</td><td>82</td><td>60</td></tr>
</table>
</body></html>
```

When read by a browser, this file is rendered into the display you see within the browser's window. The same applies to e-mail clients capable of displaying HTML content. While you, as a person, can obviously put anything you want into the file, if you want the output to look right you need to follow the rules that browsers expect.

One of those rules is that each file should contain one, and only one, HTML document. That's all of the content between the `<HTML>` tag and the `</HTML>` tag (tag names aren't case-sensitive, and it's common to see them in all-lowercase as in the example above). I mention this because one of the most common things I'll see folks do in PowerShell looks something like this:

```
Get-WmiObject -class Win32_OperatingSystem | ConvertTo-HTML | Out-File report.html
Get-WmiObject -class Win32_BIOS | ConvertTo-HTML | Out-File report.html -append
Get-WmiObject -class Win32_Service | ConvertTo-HTML | Out-File report.html -append
```

"Aaarrrggh," says my colon every time I see that. You're basically telling PowerShell to create three complete HTML documents and jam them into a single file. While some browsers (Internet Explorer, notable) will figure that out and display something, it's just

wrong. Once you start getting fancy with reports, you'll figure out pretty quickly that this approach is painful. It isn't PowerShell's fault; you're just not following the rules. Hence this guide!

You'll notice that the HTML consists of a lot of other tags, too: `<TABLE>, <TD>, <HEAD>`, and so on. Most of these are *paired*, meaning they come in an opening tag like `<TD>` and a closing tag like `</TD>`. The `<TD>` tag represents a table cell, and everything between those tags is considered the contents of that cell.

The `<HEAD>` section is important. What's inside there isn't normally visible in the browser; instead, the browser focuses on what's in the `<BODY>` section. The `<HEAD>` section provides additional meta-data, like what the title of the page will be (as displayed in the browser's window title bar or tab, not in the page itself), any style sheets or scripts that are attached to the page, and so on. We're going to do some pretty awesome stuff with the `<HEAD>` section, trust me.

You'll also notice that this HTML is pretty "clean," as opposed to, say, the HTML output by Microsoft Word. This HTML doesn't have a lot of visual information embedded in it, like colors or fonts. That's good, because it follows correct HTML practices of separating formatting information from the document structure. It's disappointing at first, because your HTML pages look really, really boring. But we're going to fix that, also.

In order to help the narrative in this book stay focused, I'm going to start with a single example. In that example, we're going to retrieve multiple bits of information about a remote computer, and format it all into a pretty, dynamic HTML report. Hopefully, you'll be able to focus on the techniques I'm showing you, and adapt those to your own specific needs.

In my example, I want the report to have five sections, each with the following information:

- Computer Information
- The computer's operating system version, build number, and service pack version.
- Hardware info: the amount of installed RAM and number of processes, along with the manufacturer and model.
- An list of all processes running on the machine.
- A list of all services which are set to start automatically, but which aren't running.
- Information about all physical network adapters in the computer. Not IP addresses, necessarily - hardware information like MAC address.

I realize this isn't a universally-interesting set of information, but these sections will allow be to demonstrate some specific techniques. Again, I'm hoping that you can adapt these to your precise needs.

Gathering the Information

I'm a big fan of modular programming. Big, big fan. With that in mind, I tend to write functions that gather the information I want to be in my report - and I'll usually do one function per major section of my report. You'll see in a bit how that's beneficial. By writing each function individually, I make it easier to use that same information in other tasks, and I make it easier to debug each one. The trick is to have each function output a single type of object that combines all of the information for that report section. I've created five functions, which I've pasted into a single script file. I'll give you each of those functions one at a time, with a brief commentary for each. Here's the first:

```
function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version;
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}
```

This is a straightforward function, and the main reason I bothered to even make it a function - as opposed to just using Get-WmiObject directly - is that I want different property names, like "OSVersion" instead of just "Version." That said, I tend to follow this exact same programming pattern for all info-retrieval functions, just to keep them consistent.

```
function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}
```

Very similar to the last one. You'll notice here that I'm using the -f formatting operator with the RAM property, so that I get a value in gigabytes with 2 decimal places. The native value is in bytes, which isn't useful for me.

```
function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}
```

Here, I've had to recognize that I'll be getting back more than one object from WMI, so I have to enumerate through them using a ForEach construct. Again, I'm primarily just renaming properties. I absolutely could have done that with a Select-Object command, but I like to keep the overall function structure similar to my other functions. Just a personal preference that helps me include fewer bugs, since I'm used to doing things this way.

```
function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}
```

Very similar to the function for services. You can probably start to see how using this same structure makes a certain amount of copy-and-paste pretty effective when I create a new function.

```
function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}
```

The main thing of note here is how I've converted the speed property, which is natively in bytes, to megabytes. Because I don't care about decimal places here (I want a whole number), casting the value as an integer, by using the -as operator, is easier for me than the -f formatting operator. Also, it gives me a chance to show you this technique!

Note that, for the purposes of this book, I'm going to be putting these functions into the same script file as the rest of my code, which actually generates the HTML. I don't normally do that. Normally, info-retrieval functions go into a script module, and I then write my HTML-generation script to load that module. Having the functions in a module makes them easier to use elsewhere, if I want to. I'm skipping the module this time just to keep things simpler for this demonstration. If you want to learn more about script modules, pick up Learn PowerShell Toolmaking in a Month of Lunches or PowerShell in Depth, both of which are available from Manning.com.

Building the HTML

I'm going to abandon the native ConvertTo-HTML cmdlet that I've discussed so far, Instead, I'm going to ask you to use the EnhancedHTML2 module that comes with this ebook. Note that, as of October 2013, this is a new version of the module - it's simpler than the EnhancedHTML module I introduced with the original edition of this book.

Let's start with the script that actually uses the module. It's included with this book as EnhancedHTML2-Demo.ps1, so herein I'm going to paste the whole thing, and then insert explanations about what each bit does. Note that I can't control how the code will line-wrap in an e-reader, so it might look wacky.

```
#requires -module EnhancedHTML2
<#
.SYNOPSIS
Generates an HTML-based system report for one or more computers.
Each computer specified will result in a separate HTML file;
specify the -Path as a folder where you want the files written.
Note that existing files will be overwritten.

.PARAMETER ComputerName
One or more computer names or IP addresses to query.

.PARAMETER Path
The path of the folder where the files should be written.

.PARAMETER CssPath
The path and filename of the CSS template to use.

.EXAMPLE
.\New-HTMLSystemReport -ComputerName ONE,TWO `
                       -Path C:\Reports\
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True,
               ValueFromPipeline=$True,
               ValueFromPipelineByPropertyName=$True)]
    [string[]]$ComputerName,

    [Parameter(Mandatory=$True)]
    [string]$Path
)
```

The above section tells us that this is an "advanced script," meaning it uses PowerShell's cmdlet binding. You can specify one or more computer names to report from, and you must specify a folder path (not a filename) in which to store the final reports.

```
BEGIN {
    Remove-Module EnhancedHTML2
    Import-Module EnhancedHTML2
}
```

The BEGIN block can technically be removed. I use this demo to test the module, so it's important that it unload any old version from memory and then re-load the revised version. In production you don't need to do the removal. In fact, PowerShell v3 and later won't require the import, either, if the module is properly located in `\Documents\WindowsPowerShell\Modules\EnhancedHTML2`.

```
PROCESS {

$style = @"
<style>
body {
    color:#333333;
    font-family:Calibri,Tahoma;
    font-size: 10pt;
}

h1 {
    text-align:center;
}

h2 {
    border-top:1px solid #666666;
}

th {
    font-weight:bold;
    color:#eeeeee;
    background-color:#333333;
    cursor:pointer;
}

.odd  { background-color:#ffffff; }

.even { background-color:#dddddd; }

.paginate_enabled_next, .paginate_enabled_previous {
    cursor:pointer;
    border:1px solid #222222;
    background-color:#dddddd;
    padding:2px;
```

```
    margin:4px;
    border-radius:2px;
}

.paginate_disabled_previous, .paginate_disabled_next {
    color:#666666;
    cursor:pointer;
    background-color:#dddddd;
    padding:2px;
    margin:4px;
    border-radius:2px;
}

.dataTables_info { margin-bottom:4px; }

.sectionheader { cursor:pointer; }

.sectionheader:hover { color:red; }

.grid { width:100% }

.red {
    color:red;
    font-weight:bold;
}
</style>
"@
```

That's called a Cascading Style Sheet, or CSS. There are a few cool things to pull out from this:

I've jammed the entire `<style></style>` section into a *here-string*, and stored that in the variable $style. That'll make it easy to refer to this later.

Notice that I've defined styling for several HTML tags, such as H1, H2, BODY, and TH. Those style definitions list the tag name without a preceding period or hash sign. Inside curly brackets, you define the style elements you care about, such as font size, text alignment, and so on. Tags like H1 and H2 already have predefined styles set by your browser, like their font size; anything you put in the CSS will override the browser defaults.

Styles also inherit. The entire body of the HTML page is contained within the `<BODY></BODY>` tags, so whatever you assign to the BODY tag in the CSS will also apply to everything in the page. My body sets a font family and a font color; H1 and H2 tags will use the same font and color.

You'll also see style definitions preceded by a period. Those are called class styles, and I made them up out of thin air. These are sort of reusable style templates that can be applied to any element within the page. The ".paginate" ones are actually used by the JavaScript I

use to create dynamic tables; I didn't like the way its Prev/Next buttons looked out of the box, so I modified my CSS to apply different styles.

Pay close attention to .odd, .even, and .red in the CSS. I totally made those up, and you'll see me use them in a bit.

```
function Get-InfoOS {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $os = Get-WmiObject -class Win32_OperatingSystem -ComputerName $ComputerName
    $props = @{'OSVersion'=$os.version
               'SPVersion'=$os.servicepackmajorversion;
               'OSBuild'=$os.buildnumber}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoCompSystem {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $cs = Get-WmiObject -class Win32_ComputerSystem -ComputerName $ComputerName
    $props = @{'Model'=$cs.model;
               'Manufacturer'=$cs.manufacturer;
               'RAM (GB)'="{0:N2}" -f ($cs.totalphysicalmemory / 1GB);
               'Sockets'=$cs.numberofprocessors;
               'Cores'=$cs.numberoflogicalprocessors}
    New-Object -TypeName PSObject -Property $props
}

function Get-InfoBadService {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $svcs = Get-WmiObject -class Win32_Service -ComputerName $ComputerName `
           -Filter "StartMode='Auto' AND State<>'Running'"
    foreach ($svc in $svcs) {
        $props = @{'ServiceName'=$svc.name;
                   'LogonAccount'=$svc.startname;
                   'DisplayName'=$svc.displayname}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoProc {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
```

```
    $procs = Get-WmiObject -class Win32_Process -ComputerName $ComputerName
    foreach ($proc in $procs) {
        $props = @{'ProcName'=$proc.name;
                   'Executable'=$proc.ExecutablePath}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoNIC {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $nics = Get-WmiObject -class Win32_NetworkAdapter -ComputerName $ComputerName `
           -Filter "PhysicalAdapter=True"
    foreach ($nic in $nics) {
        $props = @{'NICName'=$nic.servicename;
                   'Speed'=$nic.speed / 1MB -as [int];
                   'Manufacturer'=$nic.manufacturer;
                   'MACAddress'=$nic.macaddress}
        New-Object -TypeName PSObject -Property $props
    }
}

function Get-InfoDisk {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True)][string]$ComputerName
    )
    $drives = Get-WmiObject -class Win32_LogicalDisk -ComputerName $ComputerName `
           -Filter "DriveType=3"
    foreach ($drive in $drives) {
        $props = @{'Drive'=$drive.DeviceID;
                   'Size'=$drive.size / 1GB -as [int];
                   'Free'="{0:N2}" -f ($drive.freespace / 1GB);
                   'FreePct'=$drive.freespace / $drive.size * 100 -as [int]}
        New-Object -TypeName PSObject -Property $props
    }
}
```

The preceding six functions do nothing but retrieve data from a single computer (notice that their -ComputerName parameter is defined as `[string]`, accepting one value, rather than `[string[]]` which would accept multiples). If you can't figure out how these work... you probably need to step back a bit!

For formatting purposes in this book, you're seeing me use the back tick character (like after `-ComputerName $ComputerName`). That escapes the carriage return right after it, turning it into a kind of line-continuation character. I point it out because it's easy to miss, being such a tiny character.

```
foreach ($computer in $computername) {
    try {
        $everything_ok = $true
        Write-Verbose "Checking connectivity to $computer"
        Get-WmiObject -class Win32_BIOS -ComputerName $Computer -EA Stop | Out-Null
    } catch {
        Write-Warning "$computer failed"
        $everything_ok = $false
    }
```

The above kicks off the main body of my demo script. It's taking whatever computer names were passed to the script's `-ComputerName` parameter, and going through them one at a time. It's making a call to `Get-WmiObject` as a test - if this fails, I don't want to do anything with the current computer name at all. The remainder of the script only runs if that WMI call succeeds.

```
 if ($everything_ok) {
        $filepath = Join-Path -Path $Path -ChildPath "$computer.html"
```

Remember that this script's other parameter is `-Path`. I'm using `Join-Path` to combine `$Path` with a filename. `Join-Path` ensures the right number of backslashes, so that if `-Path` is "C:" or "C:\" I'll get a valid file path. The filename will be the current computer's name, followed by the .html filename extension.

```
        $params = @{'As'='List';
                    'PreContent'='<h2>OS</h2>'}
        $html_os = Get-InfoOS -ComputerName $computer |
                   ConvertTo-EnhancedHTMLFragment @params
```

Here's my first use of the EnhancedHTML2 module: The ConvertTo-EnhancedHTMLFragment. Notice what I'm doing:

1. I'm using a hashtable to define the command parameters, including both -As List and -PreContent '`<h2>OS</h2>`' as parameters and their values. This specifies a list-style output (vs. a table), preceded by the heading "OS" in the H2 style. Glance back at the CSS, and you'll see I've applied a top border to all `<H2>` element, which will help visually separate my report sections.

2. I'm running my Get-InfoOS command, passing in the current computer name. The output is being piped to...

3. ConvertTo-EnhancedHTMLFragment, which is being given my hashtable of parameters. The result will be a big string of HTML, which will be stored in $html_os.

```
        $params = @{'As'='List';
                    'PreContent'='<h2>Computer System</h2>'}
        $html_cs = Get-InfoCompSystem -ComputerName $computer |
                   ConvertTo-EnhancedHTMLFragment @params
```

That's a very similar example, for the second section of my report.

```
        $params = @{'As'='Table';
                    'PreContent'='<h2>&diams; Local Disks</h2>';
                    'EvenRowCssClass'='even';
                    'OddRowCssClass'='odd';
                    'MakeTableDynamic'=$true;
                    'TableCssClass'='grid';
                    'Properties'='Drive',
               @{n='Size(GB)';e={$_.Size}},
               @{n='Free(GB)';e={$_.Free};css={if ($_.FreePct -lt 80) { 'red' }}},
               @{n='Free(%)';e={$_.FreePct};css={if ($_.FreeePct -lt 80) { 'red' }}}}

        $html_dr = Get-InfoDisk -ComputerName $computer |
                   ConvertTo-EnhancedHTMLFragment @params
```

OK, that's a more complex example. Let's look at the parameters I'm feeding to ConvertTo-EnhancedHTMLFragment:

- As is being given Table instead of List, so this output will be in a columnar table layout (a lot like Format-Table would produce, only in HTML).
- For my section header, I've added a diamond symbol using the HTML ♦ entity. I think it looks pretty. That's all.
- Since this will be a table, I get to specify -EvenRowCssClass and -OddRowCssClass. I'm giving them the values "even" and "odd," which are the two classes (.even and .odd) I defined in my CSS. See, this is creating the link between those table rows and my CSS. Any table row "tagged" with the "odd" class will inherit the formatting of ".odd" from my CSS. You don't include the period when specifying the class names with these parameters; only the CSS puts a period in front of the class name.
- `-MakeTableDynamic` is being set to $True, which will apply the JavaScript necessary to turn this into a sortable, paginated table. This will require the final HTML to link to the necessary JavaScript file, which I'll cover when we get there.
- `-TableCssClass` is optional, but I'm using it to assign the class "grid." Again, if you peek back at the CSS, you'll see that I defined a style for ".grid," so this table will inherit those style instructions.

- Last up is the `-Properties` parameter. This works a lot like the `-Properties` parameters of `Select-Object` and `Format-Table`. The parameter accepts a comma-separated list of properties. The first, Drive, is already being produced by `Get-InfoDisk`. The next three are special: they're hashtables, creating custom columns just like Format-Table would do. Within the hashtable, you can use the following keys:
  - n (or name, or l, or label) specifies the column header - I'm using "Size(GB)," "Free(GB)", and "Free(%)" as column headers.
  - e (or expression) is a script block, which defines what the table cell will contain. Within it, you can use $_ to refer to the piped-in object. In this example, the piped-in object comes from Get-InfoDisk, so I'm referring to the object's Size, Free, and FreePct properties.
  - css (or cssClass) is also a script block. While the rest of the keys work the same as they do with Select-Object or Format-Table, css (or cssClass) is unique to ConvertTo-EnhancedHTMLFragment. It accepts a script block, which is expected to produce either a string, or nothing at all. In this case, I'm checking to see if the piped-in object's FreePct property is less than 80 or not. If it is, I output the string "red." That string will be added as a CSS class of the table cell. Remember, back in my CSS I defined the class ".red" and this is where I'm attaching that class to table cells.
  - As a note, I realize it's silly to color it red when the disk free percent is less than 80%. It's just a good example to play with. You could easily have a more complex formula, like *if ($*\.FreePct -lt 20) { 'red' } elseif ($_.FreePct -lt 40) { 'yellow' } else { 'green' }_ - that would assume you'd defined the classes ".red" and ".yellow" and ".green" in your CSS.

```
$params = @{'As'='Table';
                       'PreContent'='<h2>&diams; Processes</h2>';
                       'MakeTableDynamic'=$true;
                       'TableCssClass'='grid'}
$html_pr = Get-InfoProc -ComputerName $computer |
                          ConvertTo-EnhancedHTMLFragment @params

$params = @{'As'='Table';
                       'PreContent'='<h2>&diams; Services to Check</h2>';
                       'EvenRowCssClass'='even';
                       'OddRowCssClass'='odd';
                       'MakeHiddenSection'=$true;
                       'TableCssClass'='grid'}

 $html_sv = Get-InfoBadService -ComputerName $computer |
                           ConvertTo-EnhancedHTMLFragment @params
```

More of the same in the above two examples, with just one new parameter: -MakeHiddenSection. This will cause that section of the report to be collapsed by default, displaying only the -PreContent string. Clicking on the string will expand and collapse the report section.

Notice way back in my CSS that, for the class .sectionHeader, I set the cursor to a pointer icon, and made the section text color red when the mouse hovers over it. That helps cue the user that the section header can be clicked. The EnhancedHTML2 module always adds the CSS class "sectionheader" to the -PreContent, so by defining ".sectionheader" in your CSS, you can further style the section headers.

```
        $params = @{'As'='Table';
                    'PreContent'='<h2>&diams; NICs</h2>';
                    'EvenRowCssClass'='even';
                    'OddRowCssClass'='odd';
                    'MakeHiddenSection'=$true;
                    'TableCssClass'='grid'}
        $html_na = Get-InfoNIC -ComputerName $Computer |
                   ConvertTo-EnhancedHTMLFragment @params
```

Nothing new in the above snippet, but now we're ready to assemble the final HTML:

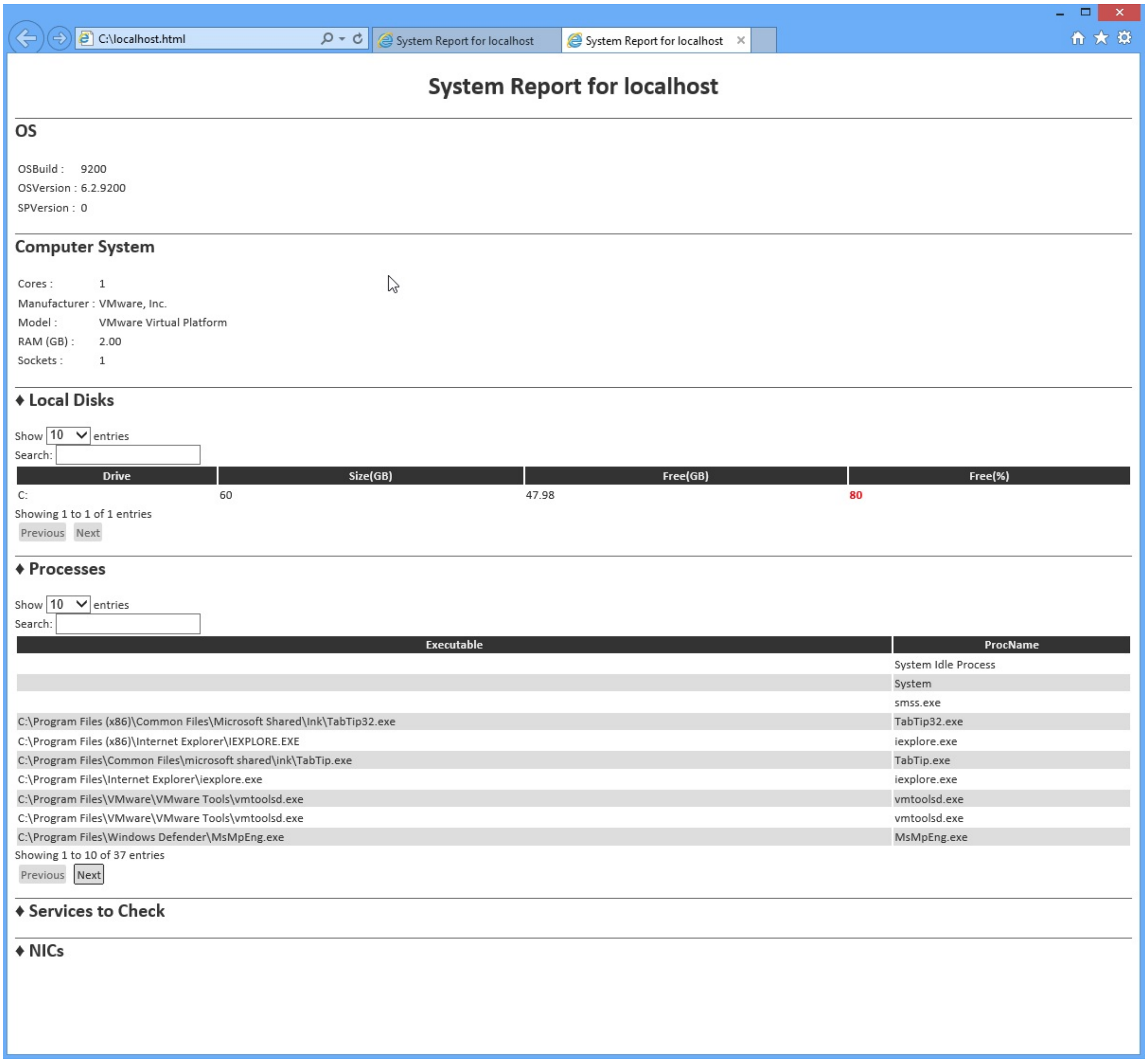
```
        $params = @{'CssStyleSheet'=$style;
                    'Title'="System Report for $computer";
                    'PreContent'="<h1>System Report for $computer</h1>";
            'HTMLFragments'=@($html_os,$html_cs,$html_dr,$html_pr,$html_sv,$html_na);
                    'jQueryDataTableUri'='C:\html\jquerydatatable.js';
                    'jQueryUri'='C:\html\jquery.js'}
        ConvertTo-EnhancedHTML @params |
        Out-File -FilePath $filepath

        <#
        $params = @{'CssStyleSheet'=$style;
                    'Title'="System Report for $computer";
                    'PreContent'="<h1>System Report for $computer</h1>";
            'HTMLFragments'=@($html_os,$html_cs,$html_dr,$html_pr,$html_sv,$html_na)}
        ConvertTo-EnhancedHTML @params |
        Out-File -FilePath $filepath
        #>
    }
}

}
```

The uncommented code and commented code both do the same thing. The first one, uncommented, sets a local file path for the two required JavaScript files. The commented one doesn't specify those parameters, so the final HTML defaults to pulling the JavaScript

from Microsoft's Web-based Content Delivery Network (CDN). In both cases:

- -CssStyleSheet specifies my CSS - I'm feeding it my predefined $style variable. You could also link to an external style sheet (there's a different parameter, -CssUri, for that), but having the style embedded in the HTML makes it more self-contained.
- -Title specifies what will be displayed in the browser title bar or tab.
- -PreContent, which I'm defining using the HTML `<H1>` tags, will appear at the tippy-top of the report. There's also a -PostContent if you want to add a footer.
- -HTMLFragments wants an array (hence my use of @() to create an array) of HTML fragments produced by ConvertTo-EnhancedHTMLFragment. I'm feeding it the 6 HTML report sections I created earlier.

The final result is piped out to the file path I created earlier. The result:



I have my two collapsed sections last. Notice that the process list is paginated, with Previous/Next buttons, and notice that my 80%-free disk is highlighted in red. The tables show 10 entries by default, but can be made larger, and they offer a built-in search box. Column headers are clickable for sorting purposes.

Frankly, I think it's pretty terrific!

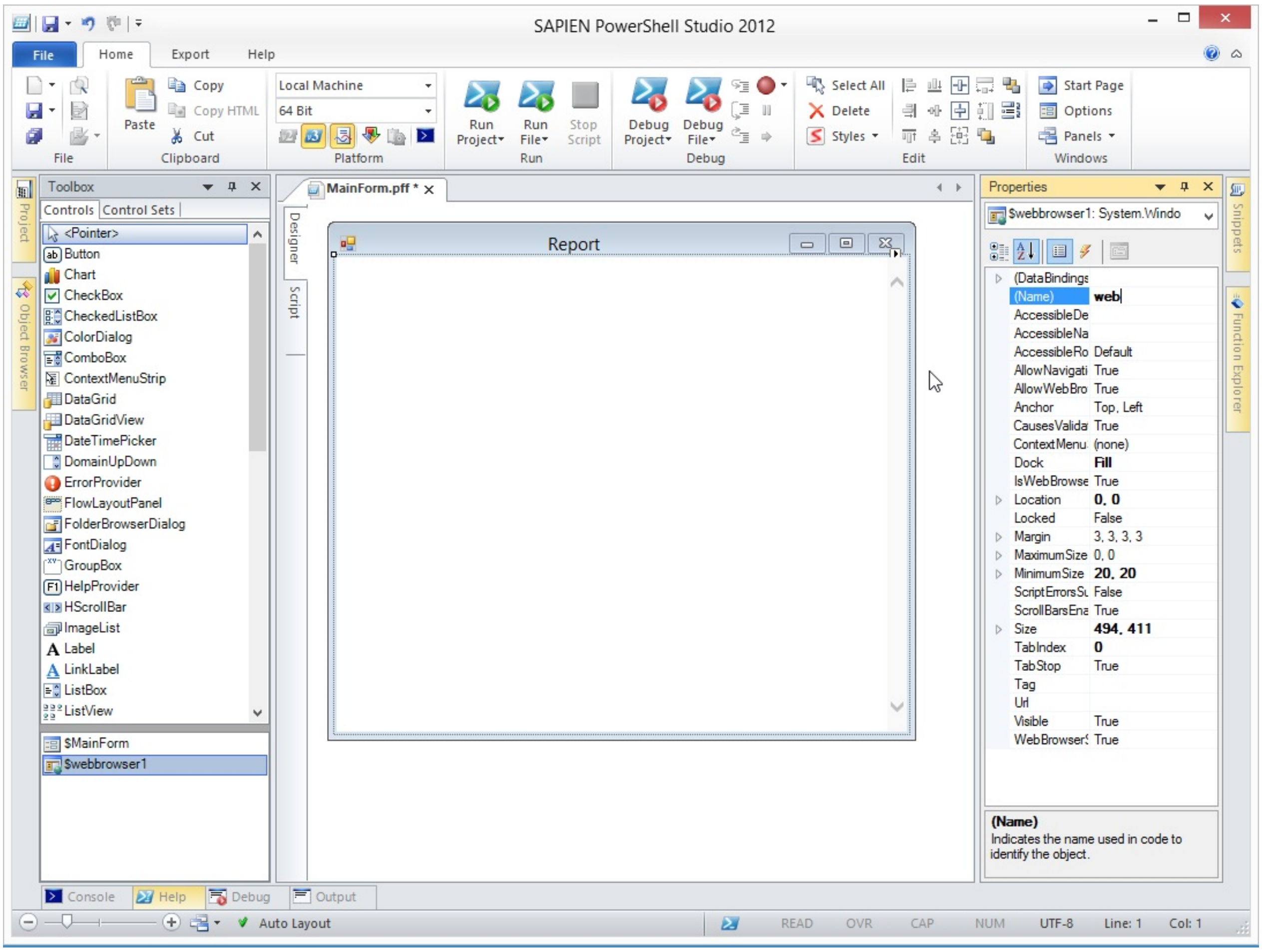
Combining HTML Reports and a GUI Application

I've had a number of folks ask questions in the forums at PowerShell.org, with the theme of "how can I use a RichTextBox in a Windows GUI application to display nicely formatted data?" My answer is don't. Use HTML instead. For example, let's say you followed the examples in the previous chapter and produced a beautiful HTML report. Keep in mind that the report stays "in memory," not in a text file, until the very end:

```
        $params = @{'CssStyleSheet'=$style;
                    'Title'="System Report for $computer";
                    'PreContent'="<h1>System Report for $computer</h1>";
                    'CssIdsToMakeDataTables'=@('tableProc','tableNIC','tableSvc');
                    'HTMLFragments'=@($html_os,$html_cs,$html_pr,$html_sv,$html_na)}
          ConvertTo-EnhancedHTML @params |
          Out-File -FilePath $filepath
```

For the sake of illustration, let's say that's now in a file named C:\Report.html. I'm going to use SAPIEN's PowerShell Studio 2012 to display that report in a GUI, rather than popping it up in a Web browser. Here, I've started a simple, single-form project. I've changed the text of the form to "Report," and I've added a WebBrowser control from the toolbox. That control automatically fills the entire form, which is perfect. I named the WebBrowser control "web," which makes it accessible from code via the variable $web.

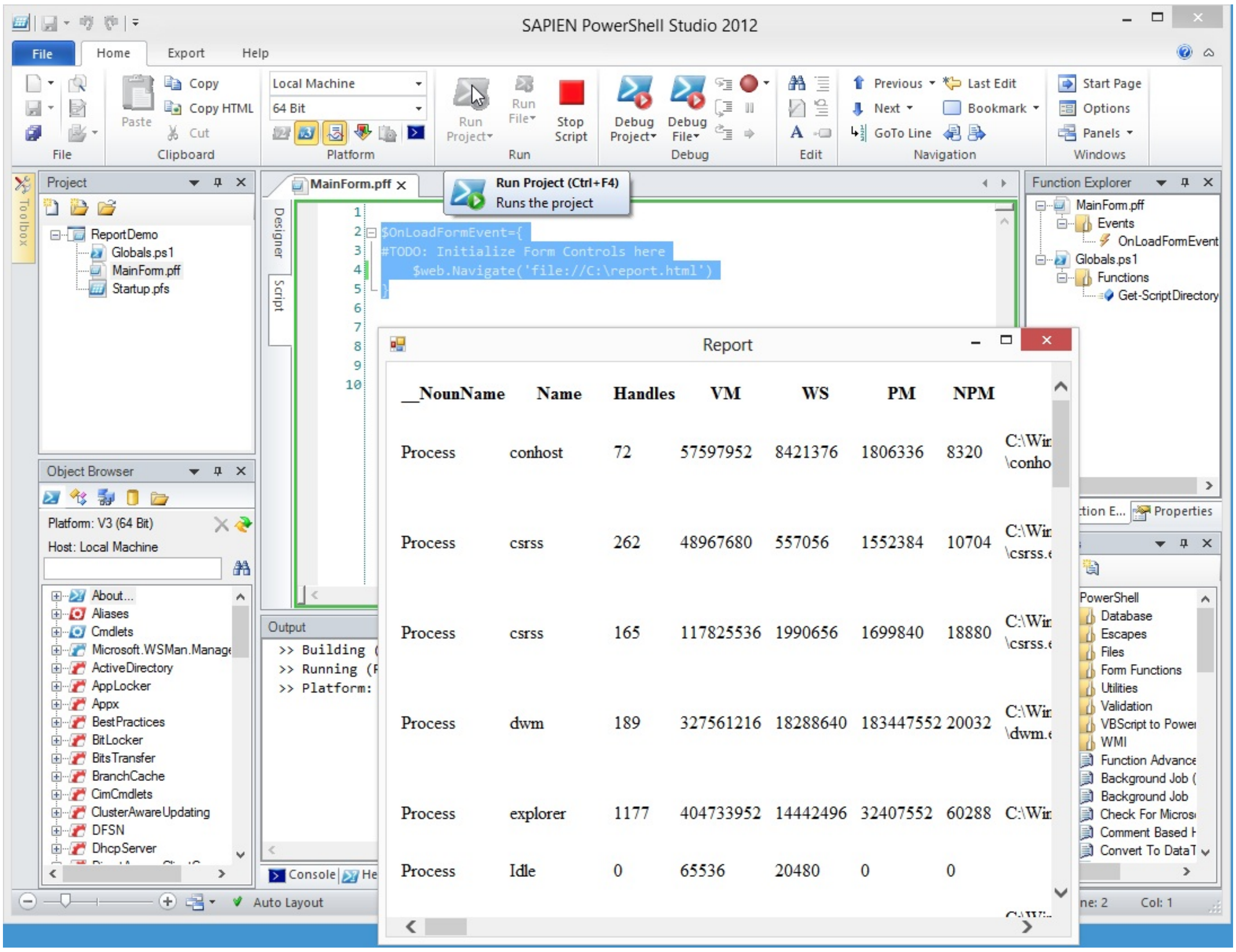
I'll note that PowerShell Studio 2012 is very out-of-date at this point, but you should still get the general idea.



I expect you'd make a form like this part of a larger overall project, but I'm just focusing on how to do this one bit. So I'll have the report load into the WebBrowser control when this form loads:

```
$OnLoadFormEvent={
#TODO: Initialize Form Controls here
    $web.Navigate('file://C:\report.html')
}
```

Now I can run the project:



I get a nice pop-up dialog that displays the HTML report. I can resize it, minimize it, maximize it, and close it using the standard buttons on the window's title bar. Easy, and it only took 5 minutes to create.

Contacting me

If you're having problems, want to do something and can't figure out how, found a bug and want to offer a correction, or just have feedback on this guide or the EnhancedHTML module, I'd love to hear from you.

The easiest way is to post in the "General Q&A" forum on http://powershell.org/wp/forums/. I keep a pretty close eye on that, and I'll respond as soon as I'm able.

Do check back from time to time, to make sure you've got the most recent version of this guide and its code.